EXHIBIT A

Disclosure BUR8-2000-0106

Created By: Sebastian Ventrone **Created On:**
Last Modified By: Cynthia Hill **Last Modified On:**

*** IBM Confidential ***

Required fields are marked with the asterisk (*) and must be filled in to complete the form .

Summary

Status	Search Results Received
Processing Location	BUR
Functional Area	DCF Signal processing, filters, signal detection, oscilators, DSPs ... 120E
Attorney/Patent Professional	Richard Henkler/Burlington/IBM
IDT Team	Tony Bonaccio/Burlington/IBM; Steve Tanghe/Burlington/IBM; Sebastian Ventrone/Burlington/IBM; Peter Mccormick/Burlington/IBM; Charles Choukalos/Burlington/IBM; Keith Williams/Burlington/IBM; Notes Intellectual Asset Development/Burlington/IBM; Joseph Iadanza/Burlington/IBM; Patricia Marmillion/Burlington/IBM
Submitted Date	1
Owning Division Select	MD
	To calculate a PVT score, use the 'Calculate PVT' button.
Incentive Program	
Lab	Beers
Technology Code	120E

Inventors with Lotus Notes IDs

Inventors: Sebastian Ventrone/Burlington/IBM, Jack Smith/Burlington/IBM

Inventor Name > denotes primary contact	Inventor Serial	Div/Dept	Manager Serial	Manager Name
> Ventrone, S.T. (Sebastian)	008381	29/RDVV	303825	Chickanosky, John D.
Smith, Jack R.	600315	29/VM8V	081511	Saiyid, Omar A.

Inventors without Lotus Notes IDs

IDT Selection

IDT Team:	Attorney/Patent Professional:
Tony Bonaccio/Burlington/IBM Steve Tanghe/Burlington/IBM Sebastian Ventrone/Burlington/IBM Peter Mccormick/Burlington/IBM Charles Choukalos/Burlington/IBM Keith Williams/Burlington/IBM Notes Intellectual Asset Development/Burlington/IBM Joseph Iadanza/Burlington/IBM Patricia Marmillion/Burlington/IBM	Richard Henkler/Burlington/IBM

Response Due to IP&L :

Main Idea

***Title of disclosure (in English)**
Performance Optimization of Code in Processor Enhanced Systems

***Idea of disclosure**

1. Describe your invention, stating the problem solved (if appropriate), and indicating the advantages of using the invention.

The rate that code is being developed for existing platforms is increasing each year. Investment in this software can often exceed the cost of the actual system. In addition, when new hardware is created, often a time lag exists from when the architecture is available and the time to migrate to the new architecture. Often, to support this legacy code, the processors keep a portion of the silicon dedicated to legacy logic that maintains backward compatibility to the prior machine instructions. The machine is placed on the market with the potential of achieving greater performance once the new software is obtained. As such, the performance gain of follow on processor architectures are often not realized by the end user.

This disclosure proposes a new hardware and software solution to more fully utilize the newer architecture features of a microprocessor. The mechanism will work real time while the program is being run by the system. The essential idea is that a fully integrated structure is included into the system architecture including the processor that will allow code translation of legacy code to enhanced code. Unlike prior attempts, this mechanism is not performed instream to the instruction fetches. It uses another approach that avoids the penalties that occur whenever real time code optimization is attempted.

At the system level, the optimization occurs as the program is being loaded into the main memory, the L3 cache, the L2 cache, and the L1 cache. While the program is running, the external optimizer is optimizing a page at a time of the legacy code. This optimization occurs even while the original non-optimized code is running. When the code is optimized, then the legacy page is replaced with the optimized page. From that point forward, the user program would see the performance boost of the optimized page.

A mechanism is built in to maintain a copy of the optimized page for future reloading of the application. This includes a mechanism to store the optimized code and processor state to a non-volatile storage device when the system is shut down. When the system is restarted, the user can choose to continue from this same state so it doesn't have to reoptimize the code. This allows a program to be optimized once and simply re-executed whenever it is used in the future.

A description of the invention is provided below.

2. How does the invention solve the problem or achieve an advantage,(a description of "the invention", including figures inline as appropriate)?

i) Start up

Upon machine start up, the operating system starts loading the requested programs from the permanent memory space, usually a hard disk area. These programs are the applications selected by the user. Assuming that no prior optimized pages exist, the program code will be non optimized. The target processor that the request is made is in this order. The processor calculates a real address, and looks up in its page table to see if the page has been generated before. If not, a new page address will be generated, and the memory load from the hard disk will occur. For optimum performance, the load will satisfied ASAP from the hard disk space to external main memory, to the L3 cache (optional), to the L2 cache, to the L1 cache, and to the dispatch unit. The processor will continue to fetch memory locations to run the selected applications, and valid not optimized pages will exists for the loaded applications. In most cases, these will be incomplete page locations in the caches, but will be a complete page in the Main memory. At this stage of operation, the machine will function in same manner as legacy machines.

ii) Instruction Optimizer

The Optimizer consists of a "processor" and associated control logic whose task is to convert non-optimized instructions into a stream of instructions that execute faster on the target processor. Collectively, the "processor" and control logic are referred to as the Instruction Recoder.

The Instruction Recoder is truly separate from the rest of the system - it may operate at a separate clock speed from the target processor and it is not affected by occurances within the main system (e.g., interrupts, exceptions). It simply reads instructions from a page of main memory, reschedules them for higher performance and stores them in a n w page in main memory. The Instruction Recoder can be implemented in two ways: 1) Microcontroller that executes recoding algorithm stored in ROM or 2) Hardwired logic

Since both the Instruction Recoder and target processor have access to main memory, this invention requires that main memory be multiported (two read ports and two write ports). This allows both processes to operate at the same time without causing structural hazards. In other words, the Instruction Recoder can read one non-optimized instruction and write one optimized instruction per cycle, and the target processor can read/write one item from this same memory element. In another embodiment, the Instruction Recoder requests a priority interrupt to the memory subsystem whenever it needs to access main memory, and it is granted access when the target processor is not using the memory bus.

The Instruction Recoder must be able to detect self-modifying code and react accordingly to it. This can be implemented by a direct communication from the target processor to the Instruction Recoder, or it can be detected by the Instruction Recoder when the L2/L3 caches write back to main memory. In any case, the detection of modified code causes the Instruction Recoder to 1) Check its optimized code to ensure it is still valid and make changes if needed OR 2) Disable optimization for the entire section of self-modifying code. In essence, the Instruction Recoder must account for any code changes after the non-optimized instructions are loaded into main memory.

The actual work of optimizing code is straightforward. The Instruction Recoder knows the architecture of the target processor, including the following things:

Number of execution units, types of units (e.g., 4 integer, 2 floating point, 2 multimedia)
Latency of each operation (e.g., 1 cycle for add, 3 cycles for multiply)
Number of architected registers, number of ports on register files and caches

Therefore it knows how much parallelism is in the target processor, how fast operations execute and the limits for getting data to/from the processor. Knowing this information, the Instruction Recoder analyzes the non-optimized instruction stream to find a "more efficient" way to schedule the instructions. "More efficient" means that the new code utilizes more resources in parallel (prevents execution units from going idle) and it has less pipeline stalls. The Instruction Recoder uses common techniques used by compilers (such as Loop Unrolling) to improve the scheduling of instructions. If the target processor has a superior architecture than the one assumed for the non-optimized code, then the Instruction Recoder should be able to improve the instruction scheduling such that the Optimized code executes faster than the Non-optimized code. In addition, because the act of optimizing code is not in the I-fetch path for the target processor, there is no penalty for creating the optimized code. In fact, this is transparent to the user until target processor starts executing the Optimized code, and then they experience the performance improvement.

iii) Mechanism to write optimized code to lower level caches (L1 and L2)

Once the optimized page has fully been updated in the main memory (In other words, a duplicate page has been created, but has been optimized for the new processor architecture), the system caches must be updated. Since these are pages in which are newly assigned, there is zero risk to running applications to preload these pages into the L3, L2, and the L1. The system logic has full access to the L3, and often the L2 caches so the preloading of these caches could follow existing protocol. Preloading the L1 cache would not be required, but for optimum performance may be desired. The mechanism to preload the L1 would be to use standard DMA cycle steal functions, where the system memory management would monitor the state of the processor function. Whenever bandwidth was available on the bus, the

system memory management system would DMAC the update page entries into the L1 cache. Again, since these are new pages, the running applications would not risk corrupting the running application.

Translation Lookaside Buffer

iv) Mechanism to update page table in Main Memory and TLB in CPU

Once a sufficient portion of the new page has been updated into the memory structure, the old legacy page is ready to be swapped out for the new optimized page. Within the processor is a Page Table Buffer, that stores a n number of active pages. These pages are stored in the TLB to reduce the numbers of time that the page translation address generation would occur. The larger the TLB, the more efficient will be the memory transactions. This invention adds a DMAC path into the TLB. the memory system already knows the page entry that was now optimized. A TLB DMAC cycles occur where the existing TLB page entry is invalidated. In the simplest case this would be sufficient to pick up the new page since the processor would now have to go to the operating system and recalculate the page address, and in this case, the assigned page address could be the new optimized page entry. The preferred embodiment, the system management hardware would preload the new optimized page entry into the processor TLB. Then validate the new page, and invalidate the legacy page. Now, when the linear to page lookup occurs, and the index address is in a safe match, or adjusted (described in Jack section) the new opimized page will valid and the processor will continue at this point.

A circuit will exist on the processor to allow up to n address to be loaded for address translation/jump to a new location within the optimized page. This buffer will be preloaded by the system memory control with the DMAC protocol or via a IO serial port as collection of IO mapped registers. The Safe transition comparator will be comparing current legacy page index with translated indexes and seeing if a stored match occurs. If match occurs, then the page swap will be allowed as described, and the new translated index will be substituted for the legacy index. In this manner the code will be functionally re-aligned and operation can continue from the new optimized page. Since the new optimized page is preloaded, the user will not see a cache miss penalty, BUT from this point on would have improved performance for this page of the applications.

v) Optional mechanism to invalidate non-optimized code in L1/L2 caches to free up space

The legacy page will now be stale date in the L1, L2 and L3 caches. Two methods could exists to free up this space. One method would be to allow the existing LRU to remove the entries. Since no further requests will occur for these pages (since the new optimized pages are now the valid translations), the data will become LRU naturally, and be removed. An alternative means could be implemented so that once a legacy page has been swapped with the optimized page, the cache control logic could pro

active update all entries in the
cache tags to look up page index (stored in the Tags), and when it recognizes a replaced tag entry (in a stored register saying last
replaced page), the LRU value would be update for the line entry to mark the legacy page as LRU. In this manner, if a legacy page
had been recently used, and some other valid data still exists for that line, the legacy line entry will be marked for replacement and not
the still valid entry. This optional portion of the invention could improve L1, and even L2 cache hit rates.

vi) Method to write CPU state and optimized code to virtual disk at shutdown

Upon shut down, additional hardware resources can be allocated to preserve the current optimized state of the user application codes.
If the system is powered down without this restoration means, then the next time the system is powered up, the user will start with the
non-optimized legacy code, and the machine learning/optimizing of the application will have to re-occur. Some additional claims of
this invention is the structure to maintain the optimized state of the processor. The valid optimized page entries contained in memory
will be stored in a new optimized page address memory. The system upon shutdown, will read the Page Entry Memory (PEM), and
write these locations of main memory into a Recovery Optimized Page Disk (ROPD). In addition the PEM contents will be saved
onto the ROPD.

Upon restart, the system will first recover the contents of the ROPD into the main memory, but not into the caches. Now, as the user
runs prior optimized code, the code will still exist in main memory, and the user will always see the optimized results, without the optimizer
having to repeat the whole page optimization routine.

vii) Method to control the speed of the optimizer (# of instructions optimized per second)
How much effort is done to optimize each instruction? Is the optimizer sufficiently ahead of the processor? If it is, then the optimizer spends more time optimizing the code. Otherwise, the optimizer reduces its effort. This is useful at startup when the processor and optimizer are working with the same instructions.

One option the user may have when loading applications is to be asked whether or not to allow the optimizer to optimizer the page before allowing
any execution of the page. While this is not the core of the invention, this could be allowed to occur for applications that are not timing critical.

Due to the unique nature of this invention, code can begin executing immediately upon the processor without the enhanced features of the processor, and then in real time, the code is shifted over to the enhanced architecture mode without the need of disrupting the service of the
machine. The user can have the choice of saving the upgrade, so that future system use will not have the initial performance penalty.